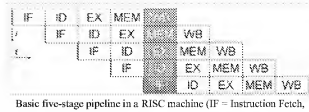
Exhibit A – Extrinsic Evidence Regarding Proper Interpretation of "Instruction Pipeline"

Instruction pipeline

From Wikipedia, the free encyclopedia

Pipelining redirects here. For HTTP pipelining, see HTTP pipelining.

An **instruction pipeline** is a technique used in the design of computers and other digital



Basic five-stage pipeline in a RISC machine (IF = Instruction Fetch, ID = Instruction Decode, EX = Execute, MEM = Memory access, WB = Register write back)

electronic devices to increase their instruction throughput (the number of instructions that can be executed in a unit of time).
Pipelining assumes that *successive instructions in a program sequence will overlap in execution*, as suggested in the next diagram (vertical 'i' instructions, horizontal 't' time).
Most modern CPUs are driven by a clock. The CPU consists internally of logic and flip flops. When the clock arrives, the flip flops take their new value and the logic then requires a period of time to decode the new values. Then the next clock pulse arrives and the flip flops again take their new values, and so on. By breaking the logic into smaller pieces and inserting flip flops between the pieces of logic, the delay before the logic gives valid outputs is reduced. In this way clock period can be reduced. For example, the RISC pipeline is broken into five stages with a set of flip flops between each stage.

1. Instruction fetch
2. Instruction decode and register fetch
3. Execute
4. Memory access
5. Register write back

Hazards: When a programmer (or compiler) writes assembly code, they make the assumption that each instruction is executed before execution of the subsequent instruction is begun. This assumption is invalidated by pipelining. When this causes a program to behave incorrectly, the situation is known as a hazard. Various techniques for resolving hazards such as forwarding and stalling exist.

A non-pipeline architecture is inefficient because some CPU components (modules) are idle while another module is active during the instruction cycle. Pipelining does not completely cancel out idle time in a CPU but making those modules work in parallel improves program execution significantly.

Processors with pipelining are organised inside into stages which can semi-independently work on separate jobs. Each stage is organised and linked into a 'chain' so each stage's output is inputted to another stage until the job is done. This organisation of the processor allows overall processing time to be significantly reduced.

Unfortunately, not all instructions are independent. In a simple pipeline, completing an instruction may require 5 stages. To operate at full performance, this pipeline will need to run 4 subsequent independent instructions while the first is completing. If 4 instructions that do not depend on the output of the first instruction are not available, the pipeline control logic must insert a stall or wasted clock cycle into the pipeline until the dependency is resolved. Fortunately, techniques such as forwarding can significantly reduce the cases where stalling is required. While pipelining can in theory increase performance over an unpipelined core by a factor of the number of stages (assuming the clock frequency also scales with the number of stages), in reality, most code does not allow for ideal execution.

Contents

Advantages and Disadvantages

Pipelining does not help in all cases. There are several disadvantages associated. An instruction pipeline is said to be *fully pipelined* if it can accept a new instruction every clock cycle. A pipeline that is not fully pipelined has wait cycles that delay the progress of the pipeline.

Advantages of pipelining:

1. The cycle time of the processor is reduced, thus increasing instruction bandwidth in most cases.

Advantages of not pipelining:

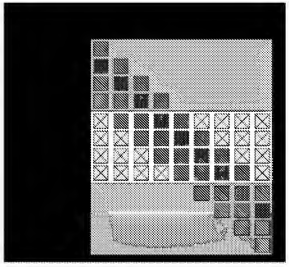
1. The processor executes only a single instruction at a time. This prevents branch delays (in effect, every branch is delayed) and problems with serial instructions being executed concurrently. Consequently the design is simpler and cheaper to manufacture.
2. The instruction latency in a non-pipelined processor is slightly lower than in a pipelined equivalent. This is due to the fact that extra flip flops must be added to the data path of a pipelined processor.
3. A non-pipelined processor will have a stable instruction bandwidth. The performance of a pipelined processor is much harder to predict and may vary more widely between different programs.

Examples

Generic pipeline

To the right is a generic 4-stage pipeline with four stages:

1. Fetch
2. Decode
3. Execute
4. Write-back



Generic 4-stage pipeline; the colored boxes represent instructions independent of each other

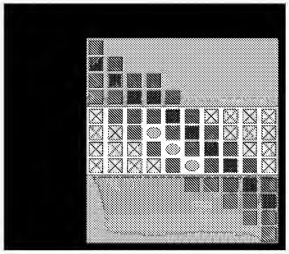
The top gray box is the list of instructions waiting to be executed; the bottom gray box is the list of instructions that have been completed; and the middle white box is the pipeline.

Execution is as follows:

Time	Execution
0	Four instructions are awaiting to be executed
1	▪ the green instruction is fetched from memory
2	▪ the green instruction is decoded ▪ the purple instruction is fetched from memory
3	▪ the green instruction is executed (actual operation is performed) ▪ the purple instruction is decoded ▪ the blue instruction is fetched
4	▪ the green instruction's results are written back to the register file or memory ▪ the purple instruction is executed ▪ the blue instruction is decoded ▪ the red instruction is fetched

5	■ the green instruction is completed ■ the purple instruction is written back ■ the blue instruction is executed ■ the red instruction is decoded
6	■ The purple instruction is completed ■ the blue instruction is written back ■ the red instruction is executed
7	■ the blue instruction is completed ■ the red instruction is written back
8	■ the red instruction is completed
9	All instructions are executed

Bubble



A bubble in cycle 3 delays execution

When a "hiccup" in execution occurs, a "bubble" is created in the pipeline in which nothing useful happens. In cycle 2, the fetching of the purple instruction is delayed and the decoding stage in cycle 3 now contains a bubble. Everything "behind" the purple instruction is delayed as well but everything "ahead" of the purple instruction continues with execution.

Clearly, when compared to the execution above the bubble yields a total execution time of 9 cycles instead of 8.

Bubbles are unlike stalls, in which nothing useful will happen for the fetch, decode, execute and writeback. It can be completed with a nop code.

Example 1

A typical instruction to add two numbers might be `ADD A, B, C`, which adds the values found in memory locations A and B, and then puts the result in memory location C. In a pipelined processor the pipeline controller would break this into a series of instructions similar to:

```
LOAD A, R1
LOAD B, R2
ADD R1, R2, R3
STORE R3, C
LOAD next instruction
```

The R locations are registers, temporary memory inside the CPU that is quick to access. The end result is the same, the numbers are added and the result placed in C, and the time taken to drive the addition to completion is no different from the non-pipelined case.

The key to understanding the advantage of pipelining is to consider what happens when this ADD function is "half-way done", at the ADD instruction for instance. At this point the circuitry responsible for loading data from memory is no longer being used, and would normally sit idle. In this case the pipeline controller fetches the next instruction from memory, and starts loading the data it needs into registers. That way when the ADD instruction is complete, the data

needed for the next ADD is already loaded and ready to go. The overall effective speed of the machine can be greatly increased because no parts of the CPU sit idle.

Each of the simple steps are usually called **pipeline stages**, in the example above the pipeline is three stages long, a loader, an adder, and a storer.

Every microprocessor manufactured today uses at least 2 stages of pipeline. (The Atmel AVR and the PIC microcontroller each have a 2 stage pipeline).

Example 2

To better visualize the concept, we can look at a theoretical 3-stages pipeline:

Stage	Description
Load	Read instruction from memory
Execute	Execute instruction
Store	Store result in memory and/or registers

and a pseudo-code assembly listing to be executed:

```
LOAD  #40, A      ; load 40 in A
MOVE  A, B        ; copy A in B
ADD   #20, B      ; add 20 to B
STORE B, 0x300    ; store B into memory cell 0x300
```

This is how it would be executed:

Clock 1

Load	Execute	Store
LOAD		

The LOAD instruction is fetched from memory.

Clock 2

Load	Execute	Store
MOVE	LOAD	

The LOAD instruction is executed, while the MOVE instruction is fetched from memory.

Clock 3

Load	Execute	Store
ADD	MOVE	LOAD

The LOAD instruction is in the Store stage, where its result (the number 40) will be stored in the register A. In the meantime, the MOVE instruction is being executed. Since it must move the contents of A into B, it must wait for the ending of the LOAD instruction.

Clock 4

Load	Execute	Store
STORE	ADD	MOVE

The STORE instruction is loaded, while the MOVE instruction is finishing off and the ADD is calculating.

And so on. Note that, sometimes, an instruction will depend on the result of another one (like our MOVE example). When more than one instruction references a particular location for an operand, either reading it (as an input) or writing it (as an output), executing those instructions in an order different from the original program order can lead to hazards
(mentioned above). There are several established techniques for either preventing hazards from occurring, or working around them if they do.

Complications

Many designs include pipelines as long as 7, 10 and even 31 stages (like in the Intel Pentium 4). The Xelerator X10q has a pipeline more than a thousand stages long [1] (http://www.mdronline.com/watch/watch_Issue.asp?Volname=Issue+%23171&on=1#item13) . The downside of a long pipeline is when a program branches, the entire pipeline must be flushed, a problem that branch predicting helps to alleviate. Branch predicting itself can end up exacerbating the problem if branches are predicted poorly. In certain applications, such as supercomputing, programs are specially written to rarely branch and so very long pipelines are ideal to speed up the computations, as long pipelines are designed to reduce clocks per instruction (CPI). Branching happens constantly, re-ordering branches such that the more likely to be needed instructions are placed into the pipeline can significantly reducing the speed losses associated with having to flush failed branches. Programs such as gcov can be used to examine how often particular branches are actually executed using a technique known as coverage analysis, however such analysis is often a last-resort for optimisation.

The higher throughput of pipelines falls short when the executed code contains many branches: the processor cannot know where to read the next instruction, and must wait for the branch instruction to finish, leaving the pipeline behind it empty. After the branch is resolved, the next instruction has to travel all the way through the pipeline before its result becomes available and the processor appears to "work" again. In the extreme case, the performance of a pipelined processor could theoretically approach that of an unpipelined processor, or even slightly worse if all but one pipeline stages are idle and a small overhead is present between stages.

Because of the instruction pipeline, code that the processor loads will not immediately execute. Due to this, updates in the code very near the current location of execution may not take effect because they are already loaded into the Prefetch Input Queue. Instruction caches make this phenomenon even worse. This is only relevant to self-modifying programs.

See also

- Wait state
- classic RISC pipeline
- Pipeline (computer)
- Parallel computing
- Branch Prediction in the Pentium Family (http://x86.org/articles/branch/branchprediction.htm)

External Links

- ArsTechnica article on pipelining (http://arstechnica.com/articles/paedia/cpu/pipelining-1.ars/1)

Retrieved from "http://en.wikipedia.org/wiki/Instruction_pipeline"

Category: Instruction processing

Jurij Šilc • Borut Robič • Theo Ungerer

Processor Architecture

From Dataflow to Superscalar and Beyond

With 132 Figures and 34 Tables

Springer

Dr. Jurij Šilc
Computer Systems Department
Jožef Stefan Institute
Jamova 39
SI-1001 Ljubljana, Slovenia

Assistant Professor
Dr. Borut Robič
Faculty of Computer and Information Science
University of Ljubljana
Tržaška cesta 25
SI-1001 Ljubljana, Slovenia

Professor Dr. Theo Ungerer
Department of Computer Design and Fault Tolerance
University of Karlsruhe
P.O. Box 6980
D-76128 Karlsruhe, Germany

Library of Congress Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek – CIP-Einheitsaufnahme

Šilc, Jurij:
Processor architecture: from dataflow to superscalar and beyond/
Jurij Šilc; Borut Robič; Theo Ungerer. - Berlin; Heidelberg; New
York; Barcelona; Hong Kong; London; Milan; Paris; Singapore;
Tokyo: Springer, 1999
ISBN 3-540-64798-8

ISBN 3-540-64798-8 Springer-Verlag Berlin Heidelberg New York

Typesetting: Camera ready pages by the authors
Cover Design: Künkel + Lopka, Werbeagentur, Heidelberg
Printed on acid-free paper SPIN 10665103 33/3142 – 5 4 3 2 1 0

- Inclusion and management of a fast-access *translation lookaside buffer* (TLB) for virtual to physical page address translation. The TLB is organized like a fully associative cache and usually contains 32 to 256 entries. Accessing the TLB takes a single machine cycle or less.
- Support of a paging mechanism involved in the virtual memory organization, for the segmentation mechanism (if implemented) and for memory protection.

The MMU access is often overlapped with the set location during cache access - a cache organization that is called virtually indexed, physically tagged. Otherwise the MMU access can be done before the cache access (a so-called *physically addressed* cache) or after cache access in the case of a cache miss (a so-called *virtually addressed* cache). *Physically tagged* caches require the cache to be compared with the physical address from the MMU. In such environments cache miss detection may be a bottleneck in the MMU. A *virtually tagged* cache uses virtual addresses when attempting to find the required word in the cache. The least significant part of the virtual address is used to access one line of the cache (direct mapped) that may contain the required word. The most significant part of the virtual address is then compared with the tag address bits for a possible match, or cache hit. This scheme ensures that cache misses are quickly detected, and that addresses are translated only on a cache miss.

For more details on caches and MMU organization, see *Hennessy and Patterson* [134] and *Shriver and Smith* [258].

1.5 Basic Pipeline Stages

One of the major features of modern processors (especially RISC processors) is the use of a pipelined instruction execution to achieve an average CPI close to 1. Pipelining is an implementation technique whereby multiple instructions are overlapped in execution. It is not visible to the programmer. Each step is called a *pipe stage* or *pipe segment*. Pipeline stages are separated by clocked *pipeline registers* (also called *latches*). A *pipeline machine cycle* is the time required to move an instruction one step down the pipeline.

Ideally, in a *k-stage pipeline* an instruction is executed in k cycles by k stages. If instruction fetching into the pipeline continues, then at any time – assuming ideal conditions – k instructions will be handled simultaneously and it will take k cycles for each instruction to leave the pipeline. We define *latency* to be the total time needed for an instruction to pass through all k stages of the pipeline. The *throughput* of the pipeline is defined to be the number of instructions that can leave a pipeline per cycle. This rate reflects the computing power of a pipeline. In contrast to the $n * k$ cycles on a hypothetical non-pipelined processor, the execution of n instructions on a k-stage pipeline will take $k + n - 1$ cycles (assuming ideal conditions

with latency k cycles and throughput 1). Hence, the resulting *speedup* is $n * k/(k + n - 1) = k/(k/n + 1 - 1/n)$. If the number of instructions that are issued to the pipeline is infinite, then the resulting speedup equals the number k of pipeline stages.

As an example of pipelined instruction execution we assume a simple instruction pipeline with the basic stages shown in Fig. 1.3. Overlapped execution of the five steps leads to a 5-stage pipeline. The pipeline execution

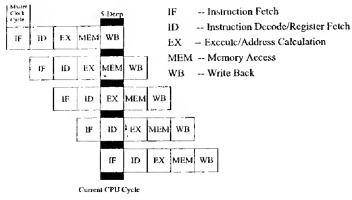


Fig. 1.3. Basic pipelining

proceeds in a smooth manner since each pipeline stage is accomplished in a single clock cycle. The RISC approach offers such a single cycle execution for most instructions. Such a pipeline can be found in the DLX[3] RISC of *Hennessy and Patterson* [134] and in the MIPS R3000 processor (Sect. 1.7.3). Today such simple RISC pipelines can be found as core pipelines in signal processors and in some multimedia processors.

Figure 1.4 shows the basic stages of the instruction pipeline in more detail. Pipeline stages are buffered by different pipeline registers:

- several *program counter registers* (PC) in the IF stage, between the IF/ID and between the ID/EX stages,
- the *instruction register* between the IF/ID stages,
- the *ALU input registers 1* and *2* and the *immediate register* between the ID/EX stages,
- the *conditional register*, the *ALU output register*, and the *store value register* between the EX/MEM stages, and
- the *load memory data register* and *ALU result register* between the MEM/WB stages.

During instruction execution the following sequence of steps is performed:

[3] DLX (pronounced "Deluxe") is a simple load/store architecture

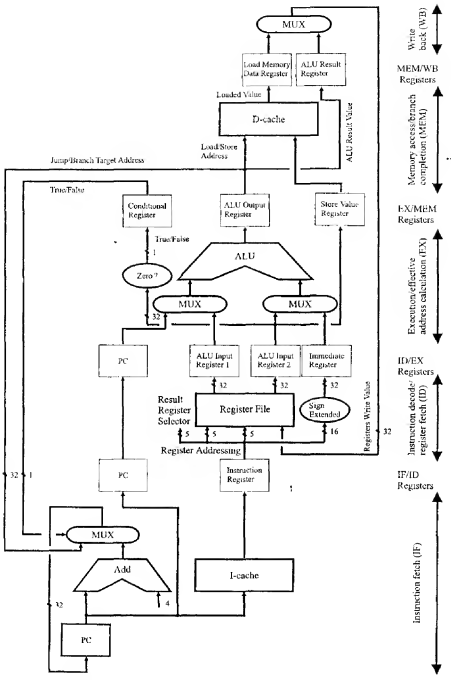


Fig. 1.4. The implementation of the DLX pipeline

H

Harvard architecture – a computer design feature where there are two separate memory units: one for instructions and the other for data.

I

I-cache – a cache that only holds the instructions of a program (not data). I-caches generally do not need a write policy.

in-order issue – the situation in which instructions are sent to be executed in the same order as they appear in the program.

instruction decoder unit – the module that receives an instruction from the instruction fetch unit, identifies the type of instruction from the opcode, assembles the complete instruction with its operands, and sends the instruction to the appropriate functional unit, or to an instruction pool to await execution.

instruction fetch unit – the module that fetches instructions from memory, usually in conjunction with a bus interface unit, and prepares them for subsequent decoding and execution by one or more functional units. If an I-cache is existent, the instructions are fetched from the I-cache.

instruction format – the specification of the number and size of all possible instruction fields in an instruction set architecture.

instruction issue – the act of initiating the performance of an instruction (not its fetch). Issue policies are important design decisions in systems that use parallelism and execution out of program order to achieve more speed.

instruction-level parallelism (ILP) – the concept of executing two or more instructions in parallel (generally instructions taken from a sequential, not parallel, stream of instructions).

instruction pipeline – a structure that separates the execution of instructions into multiple phases, and executes separate instructions in each phase simultaneously.

instruction reordering – a technique in which the CPU executes instructions in an order different from that specified by the program, with the purpose of increasing the overall execution speed of the CPU.

instruction scheduling – the relocation of independent instructions in order to maximize instruction-level parallelism (and/or minimize instruction stalls).

instruction set – the collection of all the machine-language instructions available to the programmer.

pipeline processor – a processor that executes more than one instruction at a time, in pipelined fashion. The execution of each instruction is divided into a sequence of simpler suboperations. Each suboperation is performed by a separate hardware section called a stage, and each stage passes its result to a succeeding stage. Normally, each instruction only remains at each stage for a single cycle, and each stage begins executing a new instruction as previous instructions are being completed in later stages. Thus, a new instruction can often begin during every cycle. Pipelines greatly improve the rate at which instructions can be executed, as long as there are no dependences. The efficient use of a pipeline requires that several instructions be executed in parallel, however the result of any instruction is not available for several cycles after that instruction has entered the pipeline. Thus, new instructions must not depend on the results of instructions which are still in the pipeline.

pipeline repeat rate – the number of cycles that occur between the issuance of one instruction and the issuance of the next instruction to the same functional unit.

pipeline throughput – the number of instructions that can leave a pipeline per cycle.

pipelining - splitting the CPU into a number of stages, which allows multiple instructions to be executed concurrently.

pop instruction – an instruction that retrieves contents from the top of the stack and places the contents in a specified register.

postincrementation – an addressing mode in which the address is incremented after accessing the memory value. Used to access elements of arrays in memory.

precise interrupts – an implementation of the interrupt mechanism such that the processor can restart after the interrupt at exactly where it was interrupted. All instructions that have started prior to the interrupt should appear to have completed before the interrupt takes place and all instructions after the interrupt should not appear to start until after the interrupt routine has finished.

predecrementation – an addressing mode using an index or address register in which the contents of the address are reduced by the size of the operand before the access is attempted.

prediction (of branches) – the act of guessing the likely outcome of a conditional branch decision. Prediction is an important technique for speeding up execution in overlapped processor designs. Increasing the depth of the prediction (the number of branch predictions that can be unresolved at any time) increases both the complexity and speed.

Computer Architecture A Quantitative Approach

David A. Patterson
UNIVERSITY OF CALIFORNIA AT BERKELEY

John L. Hennessy
STANFORD UNIVERSITY

With a Contribution by
David Goldberg
Xerox Palo Alto Research Center

MORGAN KAUFMANN PUBLISHERS, INC.
SAN MATEO, CALIFORNIA

Sponsoring Editor Bruce Spatz
Production Manager Shirley Jowell
Technical Writer Walker Cunningham
Text Design Gary Head
Cover Design David Lance Goines
Copy Editor Linda Medoff
Proofreader Paul Medoff
Computer Typesetting and Graphics Fifth Street Computer Services

Library of Congress Cataloging-in-Publication Data
Patterson, David A.
Computer architecture : a quantitative approach / David A.
Patterson, John L. Hennessy
p. cm.
Includes bibliographical references
ISBN 1-55860- 069-8
1. Computer architecture. I. Hennessy, John L. II. Title.
QA76.9.A73P377 1990
004.2'2--dc20 89-85227
CIP

Morgan Kaufmann Publishers, Inc.
Editorial Office: 2929 Campus Drive. San Mateo, CA 94403
Order from: P.O. Box 50490, Palo Alto, CA 94303-9953

ADVICE, PRAISE, & ERRORS: Any correspondence related to this publication or intended for the authors should be addressed to the editorial offices of Morgan Kaufmann Publishers, Inc., Dept. P&H APE. Information regarding error sightings is encouraged. Any error sightings that are accepted for correction in subsequent printings will be rewarded by the authors with a payment of $1.00 (U.S.) per correction upon availability of the new printing. Electronic mail can be sent to bugs3@vsop.stanford.edu. (Please include your full name and permanent mailing address.)

INSTRUCTOR SUPPORT: For information on classroom software and other instructor materials available to adopters, please contact the editorial offices of Morgan Kaufmann Publishers, Inc. (415) 578-9911.

Fourth Printing

6 Pipelining

6.1 What Is Pipelining?

Pipelining is an implementation technique whereby multiple instructions are overlapped in execution. Today, pipelining is the key implementation technique used to make fast CPUs.

A pipeline is like an assembly line: Each step in the pipeline completes a part of the instruction. As in a car assembly line, the work to be done in an instruction is broken into smaller pieces, each of which takes a fraction of the time needed to complete the entire instruction. Each of these steps is called a *pipe stage* or a *pipe segment*. The stages are connected one to the next to form a pipe—instructions enter at one end, are processed through the stages, and exit at the other end.

The throughput of the pipeline is determined by how often an instruction exits the pipeline. Because the pipe stages are hooked together, all the stages must be ready to proceed at the same time. The time required between moving an instruction one step down the pipeline is a machine cycle. The length of a machine cycle is determined by the time required for the slowest pipe stage (because all stages proceed at the same time). Often the machine cycle is one clock cycle (sometimes it is two, or rarely more), though the clock may have multiple phases.

The pipeline designer's goal is to balance the length of the pipeline stages. If the stages are perfectly balanced, then the time per instruction on the pipelined machine—assuming ideal conditions (i.e., no stalls)—is equal to

$$\frac{\text{Time per instruction on nonpipelined machine}}{\text{Number of pipe stages}}$$

Under these conditions, the speedup from pipelining equals the number of pipe stages. Usually, however, the stages will not be perfectly balanced; furthermore, pipelining does involve some overhead. Thus, the time per instruction on the pipelined machine will not have its minimum possible value, though it can be close (say within 10%).

Pipelining yields a reduction in the average execution time per instruction. This reduction can be obtained by decreasing the clock cycle time of the pipelined machine or by decreasing the number of clock cycles per instruction, or by both. Typically, the biggest impact is in the number of clock cycles per instruction, though the clock cycle is often shorter in a pipelined machine (especially in pipelined supercomputers). In the advanced pipelining sections of this chapter we will see how deep pipelines can be used to both decrease the clock cycle and maintain a low CPI.

Pipelining is an implementation technique that exploits parallelism among the instructions in a sequential instruction stream. It has the substantial advantage that, unlike some speedup techniques (see Chapters 7 and 10), it is not visible to the programmer. In this chapter we will first cover the concept of pipelining using DLX and a simplified version of its pipeline. We will then look at the problems pipelining introduces and the performance attainable under typical situations. Later in the chapter we will examine advanced techniques that can be used to overcome the difficulties that are encountered in pipelined machines and that may lower the performance attainable from pipelining.

We use DLX largely because its simplicity makes it easy to demonstrate the principles of pipelining. The same principles apply to more complex instruction sets, though the corresponding pipelines are more complex. We will see an example of such a pipeline in the Putting It All Together section.

6.2 The Basic Pipeline for DLX

Remember that in Chapter 5 (Section 5.3) we discussed how DLX could be implemented with five basic execution steps:

1. IF—instruction fetch
2. ID—instruction decode and register fetch
3. EX—execution and effective address calculation
4. MEM—memory access
5. WB—write back

The Winn L. Rosch Hardware Bible, Third Edition

Winn L. Rosch

SAMS
PUBLISHING
201 West 103rd Street
Indianapolis, Indiana 46290

International Standard Book Number: 1-56686-127-6

Library of Congress Catalog Card Number: 93-075014

97 96 95 10 9 8 7 6 5

Interpretation of the printing code: the rightmost double-digit number is the year of the book's printing; the rightmost single-digit, the number of the book's printing. For example, a printing code of 94-1 shows that the first printing of the book occurred in 1994.

Composed in Agaramond and MCPdigital by Macmillan Computer Publishing

Printed in the United States of America

The Arithmetic/Logic Unit

The arithmetic/logic unit handles all the decision making (the mathematical computations and logic functions) that are performed by the microprocessor. The unit takes the instructions decoded by the control unit and either carries them out directly or executes the appropriate microcode to modify the data contained in its registers. The results are passed back out of the microprocessor through the I/O unit.

Because higher clock speeds make circuit boards and integrated circuits more difficult to design and manufacture, engineers have a strong incentive to get their microprocessors to process more instructions at a given speed. Most modern microprocessor design techniques are aimed at exactly that.

One way to speed up the execution of instructions is to reduce the number of internal steps the microprocessor must take for execution. Step reduction can take two forms: making the microprocessor more complex so that steps can be combined or by making the instructions simpler so that fewer steps are required. Both approaches have been used successfully by microprocessor designers—the former as CISC microprocessors, the latter as RISC.

Another way of trimming cycles required by programs is to operate on more than one instruction simultaneously. Two approaches to processing more instructions at once are pipelining and superscalar architecture.

Pipelining

In older microprocessor designs, a chip works single-mindedly. It reads an instruction from memory, carries it out step by step, and then advances to the next instruction. Pipelining enables a microprocessor to read an instruction, start to process it, and then, before finishing with the first instruction, read another instruction. Because every instruction requires several steps each in a different part of the chip, several instructions can be worked on at once, and passed along through the chip like a bucket brigade. Intel's Pentium chips, for example, have four levels of pipelining. So up to four different instructions may be undergoing different phases of execution at the same time inside the chip.

Pipelining is very powerful, but it is also demanding. The pipeline must be carefully organized, and the parallel paths kept carefully in step. It's like a chorus singing a canon like *Frere Jacques*—one missed beat and the harmony falls apart. If one of the execution streams delays, all the rest delay as well. The demands of pipelining are one factor pushing microprocessor designers to make all instructions execute in the same number of clock cycles. Keeping the pipeline in step is easier this way.

In general, the more stages to a pipeline, the greater acceleration it can offer. But real-world programs conspire against lengthy pipelines. Nearly all programs branch. That is, their execution can take alternate paths down different instruction streams depending on the results of

calculations and decision-making. A pipeline can load up with instructions of one program branch before it discovers that another branch is the one the program is supposed to follow. In that case, the entire contents of the pipeline must be dumped, and the whole thing loaded up again. The result is a lot of logical wheel-spinning and wasted time. The bigger the pipeline, the more time wasted. The waste resulting from branching begins to outweigh the benefits of bigger pipelines in the vicinity of five stages.

Today's most powerful microprocessors are adopting a technology called *branch prediction logic* to deal with this problem. The microprocessor makes its best guess at which branch a program will take as it is filling up the pipeline. Such guesses are good enough to make pipelines of five, six, and seven stages beneficial to overall performance.

Superscalar Architectures

The steps in a program normally are listed sequentially but they don't always need to be carried out exactly in order. Just as tough problems can be broken into easier pieces, program code can be divided as well. If, for example, you want to know the larger of two rooms, you need to compute the volume of each, and then make your comparison. If you had two brains, you could compute the two volumes simultaneously. A superscalar microprocessor design does essentially that. By providing two or more execution paths for programs, it can process two or more program parts simultaneously. Of course, the chip needs enough innate intelligence to determine which problems can be split up and how to do it. The Pentium, for example, has two parallel, pipelined execution paths.

The first superscalar computer design was the Control Data Corporation 6600 mainframe, introduced in 1964. Designed specifically for intense scientific applications, the initial 6600 machines were built from eight functional units and were the fastest computers in the world at the time of their introduction.

Superscalar architecture gets its name because it goes beyond the incremental increase in speed made possible by scaling down microprocessor technology. An improvement to the scale of a microprocessor design would reduce the size of the microcircuitry on the silicon chip. The size reduction shortens the distance signals must travel and lowers the amount of heat generated by the circuit (because the elements are smaller and need less current to effect changes). Some microprocessor designs lend themselves to scaling down. Superscalar designs get a more substantial performance increase by incorporating a more dramatic change in circuit complexity.

Using pipelining and superscalar architecture cycle-saving techniques has cut the number of cycles required for the execution of a typical microprocessor instruction dramatically. Early microprocessors needed, on average, several cycles for each instruction. Many of today's chips (both CISC and RISC) actually have average instruction throughputs of less than one cycle per instruction.

Microsoft Press

Computer Dictionary

Third Edition

Microsoft Press

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Cataloging-in-Publication Data
Microsoft Press Computer Dictionary. -- 3rd ed.
p. cm.
ISBN 1-57231-446-X
1. Computers--Dictionaries. 2. Microcomputers--Dictionaries.
I. Microsoft Press.
QA76.15.M54 1997
004'.03--dc21 97-15489
CIP

Printed and bound in the United States of America.

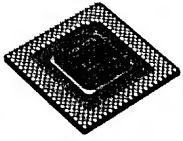
5 6 7 8 9 QMQM 2 1 0 9 8

Distributed to the book trade in Canada by Macmillan of Canada, a division of Canada Publishing Corporation.

A CIP catalogue record for this book is available from the British Library.

Microsoft Press books are available through booksellers and distributors worldwide. For further information about international editions, contact your local Microsoft Corporation office. Or contact Microsoft Press International directly at fax (425) 936-7329.

Acquisitions Editor: Kim Fryer
Project Editor: Maureen Williams Zimmerman, Anne Taussig
Technical Editors: Dail Magee Jr., Gary Nelson, Jean Ross, Jim Fuchs, John Conrow, Kurt Meyer, Robert Lyon, Roslyn Lutsch



Pin grid array. The pin grid array on the back of a Pentium chip.

pipe \pīp\ *n.* **1.** A portion of memory that can be used by one process to pass information along to another. Essentially, a pipe works like its namesake: it connects two processes so that the output of one can be used as the input to the other. *See also* input stream, output stream. **2.** The vertical line character (|) that appears on a PC keyboard as the shift character on the backslash (\) key. **3.** In UNIX, a command function that transfers the output of one command to the input of a second command.

pipeline processing \pīp´līn ¡pros`es-ēng\ *n.* A method of processing on a computer that allows fast parallel processing of data. This is accomplished by overlapping operations using a *pipe*, or a portion of memory that passes information from one process to another. *See also* parallel processing, pipe (definition 1), pipelining (definition 3).

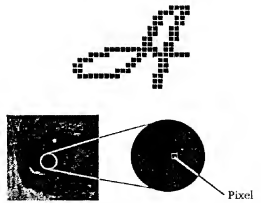
pipelining \pīp´lī`nēng\ *n.* **1.** A method of fetching and decoding instructions (preprocessing) in which, at any given time, several program instructions are in various stages of being fetched or decoded. Ideally, pipelining speeds execution time by ensuring that the microprocessor does not have to wait for instructions; when it completes execution of one instruction, the next is ready and waiting. *See also* superpipelining. **2.** In parallel processing, a method in which instructions are passed from one processing unit to another, as on an assembly line, and each unit is specialized for performing a particular type of operation. **3.** The use of pipes in passing the output of one task as input to another until a desired sequence of tasks has been carried out. *See also* pipe (definition 1), pour.

piracy \pī´ə-sē\ *n.* **1.** The theft of a computer design or program. **2.** Unauthorized distribution and use of a computer program.

.pit \dot-pit´, dot´P-I-T´\ *n.* A file extension for an archive file compressed with PackIT. *See also* PackIT.

pitch \pich\ *n.* **1.** A measure, generally used with monospace fonts, that describes the number of characters that fit in a horizontal inch. *See also* characters per inch. *Compare* point[1] (definition 1). **2.** *See* screen pitch.

pixel \piks´əl\ *n.* Short for picture (**pix**) **el**ement. One spot in a rectilinear grid of thousands of such spots that are individually "painted" to form an image produced on the screen by a computer or on paper by a printer. A pixel is the smallest element that display or print hardware and software can manipulate in creating letters, numbers, or graphics. See the illustrations. *Also called* pel.



Pixel. The letter A (top) is actually made up of a pattern of pixels in a grid, as is the cat's eye (bottom).

pixel image \piks´əl im`ə-j\ *n.* The representation of a color graphic in a computer's memory. A pixel image is similar to a bit image, which also describes a screen graphic, but a pixel image has an added dimension, sometimes called depth, that describes the number of bits in memory assigned to each on-screen pixel.

pixel map \piks´əl map`\ *n.* A data structure that describes the pixel image of a graphic, including such features as color, image, resolution, dimen-

The Indispensable PC Hardware Book

FOURTH EDITION

Hans-Peter Messmer

An imprint of Pearson Education
Boston · San Francisco · New York · Toronto · Montreal ·
London · Munich · Paris · Madrid · Cape Town ·
Tokyo · Singapore · Mexico City

PEARSON EDUCATION LIMITED

Head Office:
Edinburgh Gate
Harlow CM20 2JE
Tel: +44 (0)1279 623623
Fax: +44 (0)1279 431059

London Office:
128 Long Acre
London WC2E 9AN
Tel: +44 (0)20 7447 2000
Fax: +44 (0)20 7240 5771

Websites: www.it.minds.com
www.aw.com/cseng/

First published in Great Britain in 2002

ISBN 0 201 59616 4

British Library Cataloguing in Publication Data
A CIP catalogue record for this book can be obtained from the British Library

Library of Congress Cataloging in Publication Data
Messmer, Hans-Peter.
[PC-Handwarebuch. English]
The indispensible PC hardware book / Hans-Peter Messmer.--4th ed.
p. cm.
Includes index
ISBN 0-201-59616-4 (alk. paper)
1. Computer input-output equipment. 2. Microcomputers.

TH 78887.5 .M4613 2001
004.165--dc21

2001052081

10 9 8 7 6 5 4 3 2 1

Translated by TransScript Alba Ltd, Edinburgh, Scotland.
Typeset by Pantek Arts, Maidstone, Kent.
Printed and bound in Great Britain by Biddles Ltd of Guildford and King's Lynn.

The Publishers' policy is to use paper manufactured from sustainable forests.

Reduced instruction set and hardwired instructions

Closely related to the abbreviation RISC is the reduction of the almost unlimited instruction set of highly complex CISCs. One of the first prototypes that implemented the RISC concept, the *RISC I*, had 31 instructions, whereas its successor, the *RISC II*, had 39. The simplicity of the processor structure is shown by the reduced number of integrated transistors: in the RISC 11 there are only 41 000 (in comparison with more than 1 million in the i486 and 3 million in the Pentium). What is also interesting is that the RISC prototypes already had an additional on-chip cache, which was larger than the actual processor. In the i486 the supporting units for the processor take up more space on the processor chip than the highly efficient CPU itself.

One additional very important characteristic is that the instructions (or, put somewhat better, the hardwired Control Unit CU) are hardwired. This means that in a RISC processor, the execution unit (EU) is no longer controlled by the CU with the assistance of extensive microcodes. Instead, the whole operation is achieved in the form of hardwired logic. This greatly speeds up the execution of an instruction.

For example, in a CISC the complexity of a multiplication instruction is located in a very extensive microcode which controls the ALU. In contrast, for a RISC CPU the chip designers put the complexity in a complicated hardware multiplier. Typically, in a CISC CPU multiplications are carried out by many additions and shifts, whereas a RISC multiplier performs that operation in one or two (dependent on the precision) passes. Due to the reduced number of machine instructions, there is now enough space on the chip for implementing such highly complex types of circuit.

Instruction pipelining

As a result of the basic principles on which microprocessors work, the execution structure of an instruction is the same for the majority of machine code instructions. The following steps must be carried out:

- Read the instruction from memory (instruction fetching).
- Decode the instruction (decoding phase).
- Where necessary, fetch operand(s) (operand fetching phase).
- Execute the instruction (execution phase).
- Write back the result (write-back phase).

The instruction is decoded during the decoding phase and, in most cases, the operand addresses are determined here. In a CISC processor this instruction step is performed by the bus interface and the prefetcher as soon as there is enough space in the prefetch queue. Even the second step, the decoding of the instruction, is executed in the decoding unit prior to the instruction execution itself, thus the decoded microcode is available in the microcode queue. The remaining three steps are executed by microcode in the execution unit under the control of the CU. In normal circumstances, a single clock cycle is not sufficient, or the clock cycle must be very long, that is, the clock rate is very low.

Machine instructions are very well suited for pipelined execution. For comparison, let us look at address pipelining, which we have already met. In one complete bus cycle there are at least two very independent sequential processes: memory addressing and data transfer. Pipelined addressing now means that the addressing phase of the following bus cycle overlaps with the

data transfer phase of the current bus cycle. Application of this principle to instruction pipelining means that the above-mentioned five basic phases for successive instructions are each shifted by one stage relative to one another.

The decisive factor for the success of instruction pipelining is not that an instruction is *processed* completely within one cycle but instead that an instruction is *completed* for every cycle. What at first appears as linguistically subtle has enormous consequences. Here, each executable instruction is divided into a set number of sub-steps, such that the processor executes every sub-step in a single stage of a pipeline in one single clock cycle. This achieves the intended aim: single cycle machine instructions. This means that ideally, each machine code instruction is executed within one processor clock cycle, or, put another way, only one clock cycle per instruction is necessary, thus clocks per instruction (CPI) = 1. This is shown in Figure 8.6.

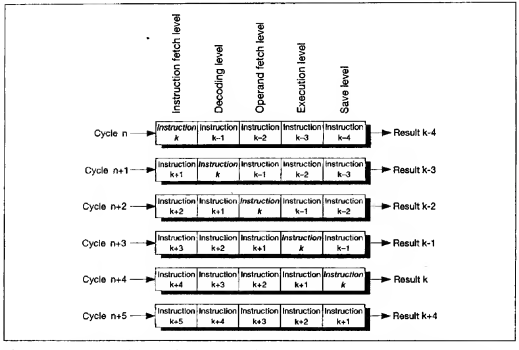


Figure 8.6: Instruction pipelining. Each instruction is split into parts in the five-level pipeline so that it can be executed. The parts are executed within one clock cycle. Therefore, for example, although instruction k requires five complete cycles for its execution, one instruction result is available for each cycle at the start of the pipeline.

As you can clearly see from the figure, the processor commences with the execution of the nth instruction as soon as the (n -1)th instruction has left the first pipeline stage. In other words, the controller unit starts the instruction fetching phase for the nth instruction as soon as the (n-1)th instruction enters the decoding phase. In this example of a five-stage pipeline, under ideal circumstances, five instructions can be found in different execution phases. It can be optimistically assumed that a processor clock cycle (PCLK) is necessary per instruction phase and, therefore, an instruction is always executed within five clock cycles. As there are five

instructions simultaneously in the pipeline, which are each displaced by one clock cycle (PCLK), an instruction result is available from the pipeline for each clock cycle (that is, each step contains an instruction in differing stages). Normally, a register is situated between the individual pipeline steps; it serves as the output register for the preceding pipeline step and, at the same time, as the input register for the following pipeline step. In comparison, without pipelining (as is normally the case with CISC processors), only the nth instruction is started, thus the instruction fetching phase of the nth instruction starts only after the (n-1)th instruction is completed – that is, after five clock cycles. Ideally, the overlapping of the instructions alone leads to increase in speed by a factor of five (!) without the need for increasing the clock rate.

The five-stage pipeline represented in Figure 8.6 is just an example. With some processors, the phases are combined into one single phase; for example, the decoding phase and the operand fetching phase (which is closely linked to the decoding phase) may be executed in a single pipeline stage. The result would be a four-stage pipeline. On the other hand, the individual instruction phases can be sub-divided even further, until each element has its own sub-phase. Thus, through simplicity, very quick pipeline stages can be implemented. This allows the clock rate to be increased. Such a strategy leads to a *superpipelined architecture* with many pipeline stages (ten or more). This superpipelined architecture allows the Alpha to achieve its speed of 300 MHz and the Pentium and PentiumPro to achieve 200 MHz.

Another possibility for increasing the performance of a RISC microprocessor is the integration of many pipelines operating in parallel. With this method, the result is a *superscalar*. One example is the Pentium with two parallel operating pipelines. I am sure you can imagine that this increases the complexity of co-ordinating the components with one another still further. Here, not only the individual pipeline stages have to co-operate but also the different pipelines themselves.

Pipeline interlocks

You can recognize one serious problem for the implementation of instruction pipelining, for example, with the two following instructions:

```
ADD eax, [ebx+ecx]

MOV edx, [eax+ecx]
```

The value of the eax register for the address calculation of the second operand in the MOV instruction is only known after the execution phase of the ADD instruction. On the other hand, the MOV instruction can already be found in the decoding phase, where the operand addresses [eax+ecx] are generated, while the MOV instruction is still in the execution phase. At this time, the decoding level decoding phase cannot determine the operand address. The CPU control unit must recognize such data and register dependencies which lead to *pipeline interlocks*, and react accordingly. The problem always appears when a following instruction n+1 (or also n+2) in an earlier pipeline stage needs the result of the instruction n from a later stage.

The simplest solution is to delay the operand calculation in the decoding phase by one clock cycle. The Berkeley RISC concept uses *scoreboarding* to deal with this pipeline obstruction. For this, a bit is attached to each processor register. For machine instructions that refer to a processor register, the bit is initially set by the control unit to show that the register value is not yet defined. The bit is removed only if the register is written to during the execution phase and its new content is valid. If a subsequent instruction wishes to use the register as an operand source, it checks whether the scoreboarding bit is set, that is, the content is undefined. If this is the case, the

Inquiry cycle
Also called snoop cycle. A bus cycle to a processor with an on-chip cache or to a cache controller to investigate whether a certain address is present in the applicable cache.

Instruction pipelining
Generally, instructions show very similar execution steps; for example, every instruction has to be fetched, decoded and executed, and the results need to be written back into the destination register. With instruction pipelining the execution of every instruction is separated into more elementary tasks. Each task is carried out by another stage of an instruction pipeline (ideally in one single clock cycle) so that, at a given time, several instructions are present in the pipeline at different stages in different execution states. Thus, not every instruction is executed completely in one clock cycle, but one instruction is completed every clock cycle.

Intel
An important US firm which manufactures microelectronic components, for example memory chips and processors. Intel is regarded as the inventor of the microprocessor.

Interlock
If a stage in a pipeline needs the result or the system element of another stage which is not yet available, this is called an interlock. Interlocks arise, for example, if when calculating a composite expression the evaluation of the partial expressions is still in progress. The requesting pipeline stage then has to wait until the other pipeline stage has completed its calculations.

Internet
A worldwide net (WAN) which initially should enable data exchange between universities and research institutes. Meanwhile, any PC user who has access to a modem and a telephone line can access the Internet.

Interrupt (software, hardware)
A software interrupt is issued by an explicit interrupt instruction INT; a hardware interrupt, however, is transmitted via an IRQ line to the processor. In both cases, the processor saves flags, instruction pointer and code segment on the stack, and calls a specific procedure, the interrupt handler.

Interrupt descriptor table
See IDT.

Interrupt descriptor table register
See IDTR.

Interrupt gate
A gate descriptor used to call an interrupt handler. Unlike a trap gate the interrupt gate clears the interrupt flag and therefore disables external interrupt requests.

Interrupt handler
See interrupt.

I/O
Abbreviation of input/output.

I/O-mapped I/O
With I/O-mapped I/O the registers of peripherals are accessed via the I/O address space, that is, ports.

PCI
Abbreviation of peripheral component interconnect. A local bus standard initiated by Intel that usually has a bus width of usually 32 bits and operates at 33 MHz. A 64-bit version is intended with the forthcoming standard 2.0. Characteristic of PCI is the decoupling of processor and expansion bus by means of a bridge. The transfer rate reaches 133 Mb/s at 32 bits and 266 Mb/s at 64 bits; bursts are carried out with any length.

PCMCIA
Abbreviation of Personal Computer Memory Card International Association. An port for credit card-sized adapters which are inserted into a PCMCIA slot.

Pentium
A powerful member of the 80x86 familiy and successor of the i486. The outstanding characteristic is the superscalar architecture with the two integer pipelines u and v. They can execute simple instructions in parallel, that is, complete two instructions within one clock cycle. An improved floating-point unit further enhances performance.

Pentium Pro
Intel's newer processor generation with 32-bit technology on which all subsequent models are based (i.e. Pentium III). The Pentium Pro integrates an L2 cache together with the CPU die in one single package. The cache runs through a dedicated L2 cache bus at the full CPU clock.

Peripheral
A device or unit located outside the system's CPU/RAM.

PGA
Abbreviation of pin grid array. A package where the terminals are provided in the form of pins at the bottom of the package.

Physical address space
The number of physically addressable bytes, determined by the number of address lines of a processor or the amount of installed memory.

PIC
Abbreviation of programmable interrupt controller. A chip used to manageme several hardware interrupts and the ordered transfer of the requests to a CPU which usually has only one input for this type of interrupt request. Thus the PIC acts as a multiplexer for hardware interrupts.

PIO
Abbreviation of programmed I/O. With PIO data is exchanged between the RAM and a peripheral not by means of DMA, but with IN and OUT instructions via the CPU.

Pipeline stage
A unit or stage within a pipeline which executes a certain partial task. A pipeline for a memory access may include the four pipeline stages address calculation, address supply, reading the value and storing the value in a register. An instruction pipeline comprises, for example, the stages instruction fetch, instruction decode, execution and register write-back.

Pipelining
Starting the execution of a function of the next cycle before the function of the current cycle has been completed. For example, the 80286 provides the address for the next read cycle in advance

of receiving the data of the current cycle. This is called address pipelining or pipelined addressing. Similarly, a processor can start the execution of parts of a complex instruction in an early pipeline stage before the preceding instruction has been completed in the last pipeline stage.

PIT
Abbreviation of programmable interval timer. A chip which outputs a pulse as soon as a programmed time period has elapsed. In the original PC designs you will find the 8253 or its successor, the 8254.

Pixel
Short form of picture element; a point on a monitor. Usually the name pixel is only used in graphics mode. The pixel may be allocated one or more bits which define the colour and brightness of the picture element.

PLA
Abbreviation of programmable logic array. A highly-integrated chip with logic gates which is used as an ASIC, and whose logic can be freely-programmed during manufacturing or by the user. A PLA usually has a field of AND gates and a field of OR gates. AND and OR can be combined to achieve any logical combinatio. This is similar to the fact that all natural numbers can be generated with 0 and 1.

PLCC
Abbreviation of plastics leaded chip carrier. A type of case where the contacts are sited on all of the four sides.

Plug&Play
A standard in which new (compliant) hardware that is connected to, or integrated in, a PC is automatically recognised, set up and configured when the PC is booted.

PMOS
Abbreviation of P-channel MOS. A technology used to manufacture MOS transistors where the channel conductivity is based on positively charged holes.

Polarization
If the electric or magnetic field of an electromagnetic wave is oscillating in one direction only, the wave is linearly polarized. The direction of the magnetic field is called the polarization direction.

Polarization filter
A device used to separate part of a specific polarization direction from an electromagnetic wave. Only that part whose polarization direction coincides with the polarization direction of the filter passes through the filter.

Port
An address in the CPU's I/O address space. Usually, registers in peripherals are accessed via ports.

Positioning time
The time period between an instruction to position the read/write head and the head being moved to the indicated track.

POST
Abbreviation of power-on self test. A program in ROM which detects and checks all installed components during power-on. You can use a specially-designed expansion card to display these codes which is especially helpful when trying to detect the source of errors.